PHYC 2050 Class Notes

Thomas J. Duck

Department of Physics and Atmospheric Science

Dalhousie University

tom.duck@dal.ca

March 17, 2010

CHAPTER

FIVE

OPTIMIZATION PROBLEMS

Now that we have a good model for the flight of a golf ball, we can use it to determine optimum values for each of the different parameters. For example, suppose that you can drive a golf ball at a 70 m/s like the pros. At what angle should the ball be shot to achieve the greatest distance? What should be the backspin rate?

Keep in mind that our model will only be able to reveal what the optimum parameters are. To learn how to launch a golf ball with those parameters you will need a pro coach, equipment suited to your stroke, etc.

5.1 Exploring the Parameter Space

It often helps to begin with a simple program that allows you to tweak parameters and see the results. The program below plots out the trajectories for a golf ball launched at three different angles:

```
#! /usr/bin/env python

# golf_trajectories.py

import numpy
import pylab, matplotlib

from numpy import pi

import cgolf

def get_U(x,z):
        return 0.

V0 = 70.
rpm0 = 3000.

# Get a series of trajectories and plot them
pylab.figure(figsize=(6,4))
for theta0 in [5,20,35]:
    x,y = cgolf.get_trajectory(V0,theta0,rpm0,get_U=get_U)
    pylab.plot(x,y,label='%.0f deg'%theta0)

pylab.xlabel('Distance (m)')
pylab.ylabel('Altitude (m)')

pylab.legend()
pylab.show()
```

Figure 5.1 gives the plot the program produces. Of the three trajectories, the one with the intermediate launch angle traveled the furthest.

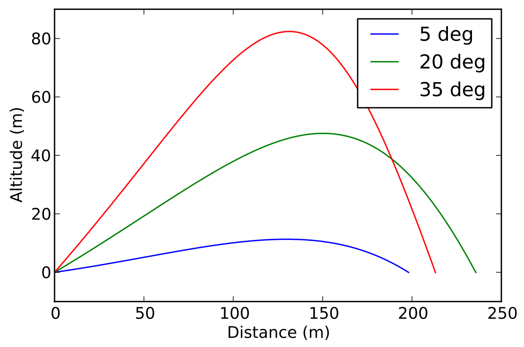

Figure 5.1: Trajectories for golf balls launched at three different angles.

What we are interested in most is the final distance. The previous program could provide three measurements of that. Instead of modifying that, however, let's calculate the distance for a range of launch angles and plot the results out in Figure 5.2:

```
#! /usr/bin/env python

# golf_angles.py

import numpy
import pylab, matplotlib

from numpy import pi

import cgolf

V0 = 70.
rpm0 = 3000.

angles = numpy.arange(1,45)
distances = []
for angle in angles:
    x,y = cgolf.get_trajectory(V0,angle,rpm0)
    distances.append(cgolf.get_total_distance(x,y))


pylab.figure(figsize=(6,4))

pylab.plot(angles,distances)

pylab.xlabel('Launch angle (deg)')
pylab.ylabel('Distance (m)')

pylab.show()
```

You can see that there is an optimum angle of nearly 15 degrees for achieving maximum distance. The program uses a spin rate of 3000 rpm, and so we might ask next how the distance varies against both lauch angle *and* speed. This will require us to fill in a table of values (like in a spreadsheet), but we don't know how to do that yet.

Suppose the angles (10–30 degrees) and rotation rates (2000–4000 rpm) are chosen by the following statements:

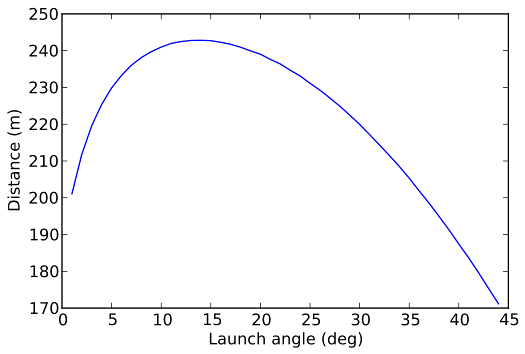

Figure 5.2: Distance the golf ball travels versus launch angle.

```
angles = numpy.arange(10,31)
rpms = numpy.arange(2000,4100,100)
```

We can allocate a 2D array of zeros with the right number of rows and columns as follows:

```
distances = numpy.zeros( [len(angles),len(rpms)] )
```

The argument is a sequence that gives the number of rows and columns, respectively. Elements in memory are accessed by using both row and column indices; e.g.:

```
distances[i,j]
```

which is similar to how we indexed one-dimensional arrays before.

We can fill in the table with a nested loop like so:

```
for i in range(len(angles)):      # Iterate through the angles
    for j in range(len(rpms)):    # Iterate through the spin rates

        # Get the current angle and spin rate
        angle = angles[i]
        rpm = rpms[i]

        # Get the trajectory for these parameters
        x,z = golf.get_trajectory(V,angle,rpm)

        # Calculate the distance from the trajectory
        distances[i,j] = golf.get_total_distance(x,z)
```

Python provides some syntactic sugar to write this more compactly:

```
for i,angle in enumerate(angles):
    for j,rpm in enumerate(rpms):
        distances[i,j] = \
            golf.get_total_distance(*golf.get_trajectory(V,angle,rpm) )
```

`enumerate()` is another built-in python function. When used in a for loop, it allows one to iterate through the indices and corresponding values of a sequence. The special *-notation used with the `get_trajectory()` function parses the tuple returned by that function and uses the values as arguments to the `get_total_distance()` function. The "\" indicates that the statement in progress is continued on the next line.

Once the distances array is filled in, we will want to plot it. 2D arrays are best presented as a contour plot, and pylab has a function for that:

```
pylab.contourf(rpms,angles,distances,50)
```

The `pylab.contourf()` function draws filled contours, and expects you to provide it with the row values, column values, 2D array values, and number of colour steps, exactly in that order. Axis labels etc are set as usual. There is also the `pylab.colorbar()` function to give colours that are used to fill in the contours meaning.

Below is the code listing for the program, and Figure 5.3 gives the plot.

```
#! /usr/bin/env python

# golf_parameter_space.py

import numpy
import pylab, matplotlib

import cgolf

V0 = 70.

# Set up a range of angles and spin rates
angles = numpy.arange(10,31)
rpms = numpy.arange(2000,4100,100)

# Create a 2D array to hold the total distance values
distances = numpy.zeros([len(angles),len(rpms)])

# Calculate the distance for each combination of angles and rpms
for i,angle in enumerate(angles):
    for j,rpm in enumerate(rpms):
        distances[i,j] = \
            cgolf.get_total_distance(*cgolf.get_trajectory(V0,angle,rpm) )


# Contour plotting
pylab.figure(figsize=(6.,5))
pylab.contourf(rpms,angles,distances,50)

colorbar = pylab.colorbar(format='%.0f',orientation='horizontal',
                          ticks=matplotlib.ticker.MaxNLocator(5))
colorbar.set_label('Distance (m)')

pylab.xlim(2000,4000)
pylab.ylim(10,30)

pylab.xlabel('Spin rate (rpm)')
pylab.ylabel('Launch angle (degrees)')
pylab.title( 'V0 = %.0f m/s'%(V0) )


# Display the plot
pylab.show()
```

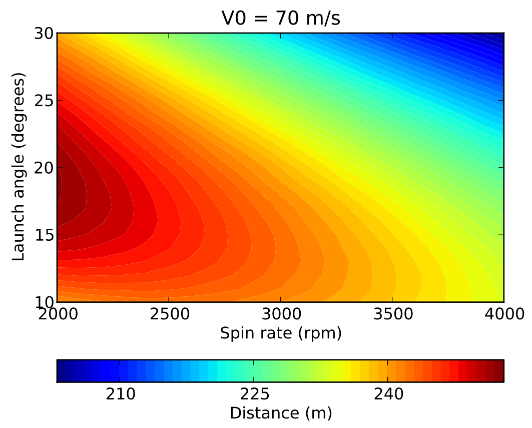

Figure 5.3: Total drive distance against launch angle and spin rate.

The plots reveals that for a launch speed of 70 m/s low spin rates are best. This is something that pro golfers can control well, although even for them it is difficult to get much below 2000 rpm. If you can achieve low spin rates, then high launch angles are best: between 15 and 20 degrees. If the spin rate is higher than 2000 rpm, then the launch angle should be lowered.

What about other launch speeds? We might continue to explore the parameter space by calculating the distance against launch angle, spin rate, and *speed*. This is probably inadvisable, for a few reasons: the resulting data will be 4-dimensional, and we won't have a good way to plot it; the calculation will require a 3D array which will require a lot of memory which we may not have; and the calculation may take a lot of time to complete.

What we are really interested in, however, is determining the optimum launch angle against spin rate and speed. We need to be able to efficiently pick off the best angle for each spin rate in a plot like Figure 5.3, and do the same thing to plots for other launch speeds. This kind of task is called an "optimization problem", because we need to determine the optimum angle against a variety of different parameters. Fortunately, scipy has some functions that can help us out.

5.2 The Brent Method

Optimizing parameters generally requires finding the maximum of some function. Suppose we have an arbitrary continuous function of one parameter, $f(x)$, that may be computed given the value `x`. Imagine that the curve is as drawn in Figure 5.4. How can we quickly find the location on the x-axis corresponding to the maximum on the curve? Put another way, how can we optimize x to produce maximum f?

We first choose three points along the x-axis, a, b and c such that $f_a < f_b > f_c$; i.e., a maximum exists somewhere between $x = a$ and $x = c$ (see Figure 5.4). The x-values a, b, and c are called *bracket points*. We want to reduce the size of the bracket until the maximum is found.

Next, choose a point b' in the larger of the two intervals [x_a, x_b] and [x_b, x_c] and calculate $f_{b'}$. Suppose that b' is in the second of the two intervals (see Figure 5.5). If $f_{b'} > f_b$ then the new bracket points around the maximum are b, b', and c. Otherwise, they are a, b, and b'. Using the new bracket points, (lather, rinse) repeat until the bracket is smaller than some acceptable tolerance. The maximum is taken to be at the middle of the three bracket points.

If the new points are chosen such that intervals are divided according to the golden ratio $((1 + \sqrt{5})/2$, then this is called the Brent Method.

The `brent()` function in `scipy.optimize` will locate the *minimum* of a function of a single variable. This is opposite to what we will most often need, but won't present a problem for us: taking the negative of a function turns

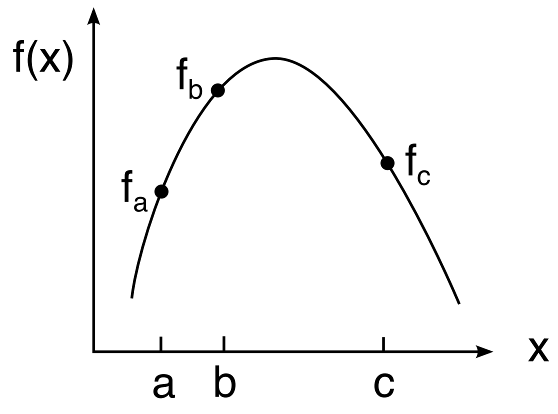

Figure 5.4: A function with three points that bracket the maximum. Note that with $f_a < f_b > f_c$ there is guaranteed to be a maximum between $x = a$ and $x = c$.

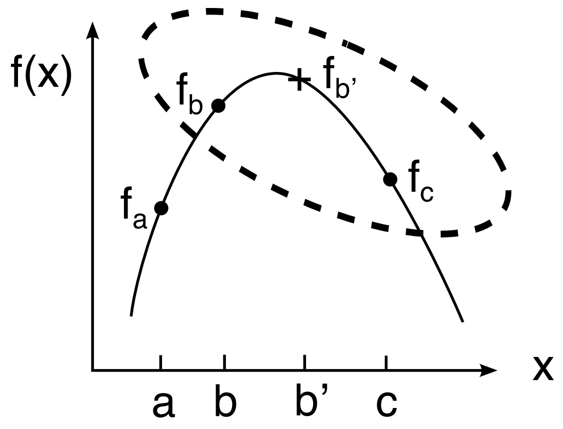

Figure 5.5: First step in the algorithm for finding a maximum. A new point b', is chosen, and the function is evaluated there. The new set of bracket points are selected (and circled) that surround the maximum.

a maximum into a minimum. `scipy.optimiz.brent()` also allows you to provide only two bracket points, in which case a downhill search is performed for a third backet point which satisfies the algorithm's criteria.

The zeros of a function can be determined using a similar approach. The `brentq()` function in `scipy.optimize` was introduced earlier for this purpose.

5.3 Optimizing the Launch Angle in Still Conditions

Let's try a simple problem first: Find the angle that gives the maximum distance from Figure 5.2. Inspection of the figure suggests that it should be around 15 degrees, and certainly somewhere between 5 and 25 degrees. The distance associate with that is about 240 m.

Here is the program that uses `scipy.optimize.brent()` to optimize the angle for maximum distance:

```
#! /usr/bin/env python

# golf_angles_findmax.py

import scipy.optimize
import cgolf

V0 = 70.
rpm0 = 3000.

# Define the function of one variable that we want to find the minimum of
def opt_func(angle):
    return -cgolf.get_total_distance(*cgolf.get_trajectory(V0,angle,rpm0))

# Perform the optimization
angle_max = scipy.optimize.brent(opt_func,brack=[5,15,25])

# Retrieve the maximum distance
distance = cgolf.get_total_distance(*cgolf.get_trajectory(V0,angle_max,rpm0))

print 'The optimum angle of %.1f deg gives a total distance of %.0f m' % \
    (angle_max,distance)
```

We first create a function of one variable that has a minimum. Our quantity of interest is the maximum, so we just take the negative of that. The function is provided to `scipy.optimize.brent()` along with our bracket, and it returns the optimum angle. Execution of the program gives an optimum angle of 14.1 degrees for a total distance of 243 m. These values are consistent with our estimates from earlier.

Now onto the more difficult problem. Let's determine the optimum angle for all the different combinations of launch speeds and spin rates. This will require another table of values, and so will result in a contour plot. The program is given below. Notice that we define the tolerance for the optimum angle so that the `brent()` routine doesn't have to narrow it down too closely.

```
#! /usr/bin/env python

# golf_angle_optimization.py

import numpy
import pylab, matplotlib
import scipy.optimize

import cgolf
```

```
cgolf.h = 0.01

Vs = numpy.arange(45.,82,2)
rpms = numpy.arange(2000.,4100,100)
angles = numpy.zeros([len(Vs),len(rpms)])


# Optimize the angle for all of the different launch speeds
for i,V in enumerate(Vs):
    for j,rpm in enumerate(rpms):
        angles[i,j] = scipy.optimize.brent(
            lambda theta: -cgolf.get_total_distance(
                *cgolf.get_trajectory(V,theta,rpm)), brack=[1,15,40],
            tol=0.01)

# Contour plotting
pylab.figure(figsize=(6.,5))
pylab.contourf(rpms,Vs,angles,50)

colorbar = pylab.colorbar(format='%.0f',orientation='horizontal',
                          ticks=matplotlib.ticker.MaxNLocator(5),
                          shrink=0.8)
colorbar.set_label('Optimum angle (deg)')

pylab.xlim(2000,4000)
pylab.ylim(45,80)

pylab.xlabel('Spin rate (rpm)')
pylab.ylabel('Launch speed (m/s)')

# Display the plot
pylab.show()
```

The program uses yet another built-in python function, `zip()`, which takes two or more sequences and pairs up elements with common indices. Put another way, it "zips up" the sequences like a zipper. This is best illustrated with an example:

```
>>> x = [1, 2, 3]
>>> y = ['a', 'b', 'c']
>>> z = zip(x,y)
>>> print z
[(1, 'a'), (2, 'b'), (3, 'c')]
```

If you have truly understood the *-notation from earlier, then you will also be able to understand why the following demonstrates the unzip operation:

```
>>> zip(*z)
[(1, 2, 3), ('a', 'b', 'c')]
```

Continuing on, Figure 5.6 gives the result of the calculation.

The plot seems to have a lot of noise in it, and this can be reduced by lowering the time step size `h`. We can, however, take a number of conclusions from the plot.

The average golfer (45 m/s, 3000 rpm) should be launching the ball at nearly 20 degrees elevation angle. It is probably hard to learn how to swing harder, so this golfer would be best advised to reduce the spin of their ball (which we know from before gives greater distance) and try for a high launch angle of 22 degrees.

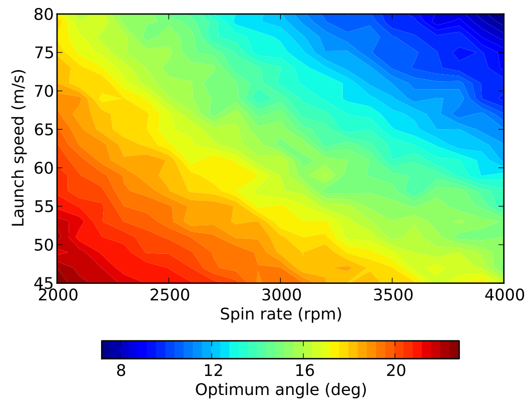

Figure 5.6: Optimum drive angle against launch speed and spin rate.

The pros who hit the ball above 70 m/s, however, should never be launching the ball any higher than about 18 degrees elevation. The pro should also keep the backspin low. If they cannot, then the launch angle will need to be reduced, maybe to even below 10 degrees elevation.

In any case, the plot shows that amateurs should be launching the ball quite a bit higher than the pros.

5.4 Optimizing the Launch Angle in Windy Conditions

How should the launch angle be changed for a strong tail wind? We can determine this by finding the difference in the optimum launch angle for two different wind speeds (say, 10 m/s and 0 m/s) in a repeat of the last experiment. Here is the code:

```
#! /usr/bin/env python

# golf_angle_optimization_wind.py

import numpy
import pylab, matplotlib
import scipy.optimize

import cgolf

cgolf.h = 0.001

Vs = numpy.arange(45.,82,2)
rpms = numpy.arange(2000.,4100,100)
angles0 = numpy.zeros([len(Vs),len(rpms)])
angles10 = numpy.zeros([len(Vs),len(rpms)])

def get_U(x,z):
    return U

for U,angles in zip([0.,10.],[angles0,angles10]):
    for i,V in enumerate(Vs):
        for j,rpm in enumerate(rpms):
            angles[i,j] = scipy.optimize.brent(
                lambda theta: -cgolf.get_total_distance(
                    *cgolf.get_trajectory(V,theta,rpm,get_U=get_U)),
                brack=[1,5],
                tol=0.01)

angles = (angles10-angles0).clip(6,10)

# Contour plotting
pylab.figure(figsize=(6.,5))
pylab.contourf(rpms,Vs,angles,50)

colorbar = pylab.colorbar(format='%.0f',orientation='horizontal',
                          ticks=matplotlib.ticker.MaxNLocator(5),
                          shrink=0.8)
colorbar.set_label('Optimum angle difference (deg)')

pylab.xlim(2000,4000)
pylab.ylim(45,80)

pylab.xlabel('Spin rate (rpm)')
pylab.ylabel('Launch speed (m/s)')

# Display the plot
pylab.show()
```

The plot given in Figure 5.7 shows that regardless of how hard the ball is hit, the launch angle in a tail wind should be increased by the same amount. For 10 m/s wind and low backspin rates the angle should be raised by about 7 degrees, and for high spin rates it is nearly 10 degrees. So, the golfer needs to adapt their stroke considerably to the conditions if maximum drive distance is to be obtained.

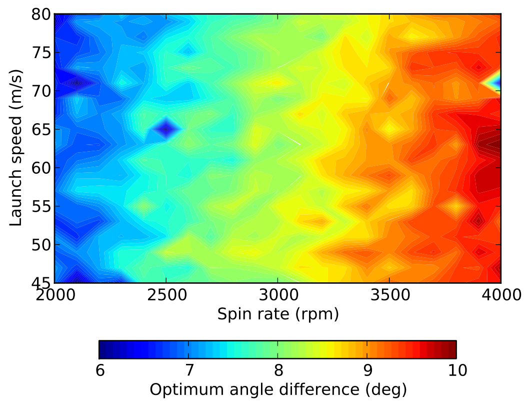

Figure 5.7: The angle change required to obtain maximum distance when there is a tail wind of 10 m/s, as compared to still conditions.

You can see there is even more noise in the plot, despite the fact we have reduce the step size. We can continue to reduce the step size, but this will result in the simulations taking longer and longer to complete. It may in fact be better to increase the error order, which is something we will explore in a later chapter.